Process Migration in the Sprite Operating System*

Fred Douglis

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

February 11, 1987

Abstract

This paper describes a process migration facility for the Sprite operating system. In order to provide location-transparent remote execution, Sprite associates with each process a distinguished *home node*, which provides kernel services to the process throughout the process's lifetime. System calls that depend on the location of a process are forwarded to the process's home node. Performance measurements based on a few simple benchmarks show that remote execution using the home-node model is efficient as long as the number of system calls that must be forwarded home is small; this appears to be the case as long as file-system-related calls can be handled without involving the home node. The benchmarks also show that the cost of migrating a process can vary from a fraction of a second to many seconds; it is determined primarily by the number of dirty virtual memory pages and file blocks associated with the process.

*This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE
11 FEB 1987

2. REPORT TYPE

3. DATES COVERED
00-00-1987 to 00-00-1987

4. TITLE AND SUBTITLE
Process Migration in the Sprite Operating System

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This paper describes a process migration facility for the Sprite operating system. In order to provide location-transparent remote execution, Sprite associates with each process a distinguished home node, which provides kernel services to the process throughout the process's lifetime. System calls that depend on the location of a process are forwarded to the process's home node. Performance measurements based on a few simple benchmarks show that remote execution using the home-node model is efficient as long as the number of system calls that must be forwarded home is small; this appears to be the case as long as file-system-related calls can be handled without involving the home node. The benchmarks also show that the cost of migrating a process can vary from a fraction of a second to many seconds; it is determined primarily by the number of dirty virtual memory pages and file blocks associated with the process.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**20**	

1 Introduction

Process migration is a method by which executing processes may be transferred between nodes in a distributed system. This paper describes a prototype process migration facility for the Sprite operating system [4,10]. The main purpose of process migration in Sprite is to make compute power easily sharable in a network of personal workstations: users with several relatively-independent tasks to perform can (potentially) complete them more quickly by offloading some of them to other workstations. For example, when recompiling a large system, the separate compilations of individual modules could be migrated to idle workstations around the network and executed in parallel. Or, simulations requiring several independent runs with varying parameters could be handled by doing each run on a separate machine.

We addressed three overall issues in implementing a process migration facility for Sprite: transparency, preemption, and efficiency. Transparency means that the results produced by a process should not depend on where it executes in the network, nor should the process need to be coded in a special way in order for it to be migrated. This means that processes must still be able to access files, devices, and other system facilities such as time-of-day and the display in the same fashion when migrated as they would if executing locally. Sprite achieves transparency by assigning each process a *home node*. Whenever a migrated process invokes a location-dependent system call (one that might produce different results if executed on different nodes), the system call is forwarded to the process's home node using a remote-procedure-call facility. Sprite's use of home nodes distinguishes it from other process migration mechanisms such as those in the V-System [2,7,8], LOCUS [5,9], and Demos/MP [6].

Our second major concern was preemption. In workstation-based computing facilities, workstations "belong" to individuals who often want and need exclusive use of their machines. We felt that a user returning to his/her node should be able to eject all of the processes that were migrated to that node in the user's absence. Remote execution facilities, such as the *rsh* facility of 4.3BSD UNIX[†] [3], permit processes to be created on remote nodes but do not allow the processes to be moved once initiated. Thus a returning user would be forced either to kill

[†] UNIX is a trademark of AT&T Bell Laboratories.

the foreign processes on his/her node or suffer degraded response until the foreign processes completed. In contrast, Sprite's process migration mechanism allows processes to be moved at any time. If a process is migrated to a node and then the node's owner resumes activity, the migrated process can be preempted from that node; after being migrated elsewhere (either home or to a different remote node), no residual dependencies are left on the node from which it was preempted.

The third major issue in implementing process migration was efficiency. In Sprite there are two potential sources of inefficiency: the time necessary to migrate a process, and the additional cost of supporting remote execution. We used a few simple benchmark programs to measure these factors in a prototype implementation of the Sprite process migration mechanism on Sun-2 workstations. Migration time is dominated by the cost of transferring the process's address space: the "null" process can be migrated in about .5 seconds, but processes with many dirty pages can require many seconds to migrate (about 130 kbytes per second). Thus, process migration makes the most sense for either (a) processes that are too "young" to have generated a large address space, or (b) processes that will execute for many seconds or minutes once migrated.

The additional cost of remote execution consists of the time required to forward system calls back to the home node. Since forwarding is relatively expensive (approximately 7 milliseconds per call), it is important that frequently-executed system calls not have to be forwarded. Sprite allows file-system-related calls to be handled without forwarding, while most other system calls must be forwarded. Since most system calls are file-system-related, remote execution appears to suffer negligible degradation: for example, a remote compilation of four files ran 1% more slowly than when performed locally.

Overall, process migration in Sprite may reduce turnaround time substantially. By distributing compilations across idle workstations, the total time to compile varying sets of programs has been reduced to 40-60% of the time needed to compile all the programs on the same host. The amount of parallelism obtained determines the reduction in turnaround time.

Section 2 of the paper discusses previous implementations of process migration. Section 3 provides background information on the Sprite system. Section 4 describes the Sprite process migration mechanism. Section 5 analyzes the cost of migration, and Section 6 measures the performance degradations suffered by migrated processes. Section 7 shows that process migration can be used to provide substantial speed-ups for some applications. Section 8 gives criteria for deciding when and where to migrate processes, based on our evaluation of the system. This paper does not address the policy decisions involved in process migration: when to migrate a process, where it should be migrated, and when to preempt a migrated process from its current remote node. We expect to begin work in these areas in the near future.

2 Related Work

Several other systems have implemented process migration in one form or another. The best-known of these are the V-System [2,7,8], LOCUS [5], and Demos/MP [6]. This section discusses how those systems address the issues of transparency, preemption, and efficiency.

All three systems provide location-transparent remote execution. Demos/MP and V are message-based systems: all interactions between a process and the rest of the world (including both the system and other user processes) are carried out by sending and receiving messages. The message system already provides a degree of location transparency in these systems; all that is needed to support transparent remote execution is a facility for forwarding messages. In Demos/MP, messages are sent to *links*, which are message paths managed by the kernel. Each link contains a globally-unique identifier for the recipient of the messages, with the identifier for the node on which the process was created, a unique local identifier for the process within that node, and the last-known location of the process. Messages are sent to the last-known location. When a process moves, its previous node forwards messages sent to the process and also notifies the senders so that they can update the link with the process's new location.

In the V-System, messages are addressed to processes, using global process identifiers. A global process identifier indicates a *logical host* for the process, which is mapped to a physical node using a cache of mappings maintained by the kernel. If a message to a process fails because the process is no longer at the same location, the kernel sending the message broadcasts to obtain the logical host's new location; this information is then used to update the cache.

In LOCUS, user process use kernel calls, rather than messages, for interaction with the system and other user processes. Thus, transparent remote execution cannot be achieved merely by forwarding messages; call-specific techniques are used. LOCUS already provides a network-transparent file system, so no special effort is needed for the file-system-related kernel calls. For kernel calls that operate on processes, the call must be forwarded to the machine on which the process is executing. Processes are named with the identifier for the node on which the process was created (called its *origin site*), plus a process identifier within the origin site. The origin site keeps track of a process's current location; other machines communicate with the origin site to find out the current location for call-forwarding.

All three of the systems allow for preemption: any process may be migrated at any time. However, if a process migrates several times then Demos/MP leaves residual dependencies at each of the process's previous nodes: each one must continue to forward messages on behalf of the process. In LOCUS only the process's origin site must assist in forwarding messages, and in V there are no residual dependencies anywhere, since the process can be located by broadcasting.

In terms of efficiency, both Demos/MP and V keep caches of a process's current location (the logical-to-physical host map kept by the V kernels, and the last-known site kept in Demos/MP links); the caches eventually get updated after a process moves, which reduces the amount of forwarding that must occur. In LOCUS there are no caches, so communication with a non-local process must always involve the process's origin site.

The V-System performs address-space *pre-copying*, an additional performance enhancement that is intended to reduce the total time during which migrated processes are suspended. Demos/MP and LOCUS both halt a process's execution while it is being migrated; the process remains frozen while its address space is copied to the new machine. In contrast, V copies the address space while the process is still running: all pages are sent to the new node, and then any pages that were modified after being sent are copied to the new node another time. This procedure repeats until the number of modified pages is small, and then the process is frozen and the rest of its state is transferred, along with any remaining dirty pages. Theimer reports that pre-copying reduces the suspension time to a period comparable to the time required to swap a process into memory; migration therefore interferes minimally with a process's interaction with other processes and the user [7].

3 The Sprite Environment

This section provides background on the Sprite system, particularly those aspects that influenced process migration. Sprite provides a kernel interface much like the UNIX operating system. To user processes, it does not appear message-based: Sprite provides a set of procedure-like system calls, consisting mostly of calls to manipulate files (open, close, read, etc.) and calls to manipulate processes (fork, exec, signal, etc.). Like UNIX, I/O devices and other special features are handled uniformly through the file system. Similarly, most interprocess communication is accomplished through the file system (though closely-coupled processes may use shared memory for communication).

Although the Sprite interface appears like that of UNIX, the implementation of the kernel is completely different. The most important feature for process migration is the availability in Sprite of a kernel-to-kernel remote procedure call (RPC) facility. The RPC mechanism allows the kernel of any node in the network to invoke services in the kernel of any other node with relatively low latency (about 5 milliseconds round-trip latency on Sun-2 workstations) and "at-most-once" semantics [10].

The RPC facility has already been used to construct a distributed file system that provides location-transparent file access across the network [11]. As in LOCUS, the network file system allowed us to make the file-system-related kernel calls location-transparent with very little additional work. For example, the basic information kept by a client for a file consists only of a token for the file (which is used to communicate with the file's server) and the current access position. This information is migrated with a process. Clients also maintain caches of recently-used file data, but the file system already contains mechanisms for ensuring cache consistency when multiple nodes access the same file; these same mechanisms are applied when a process migrates to ensure that it does not access stale data.

In Sprite, the network file system is also used as a backing store for the virtual memory system [4]. Corresponding to each segment is a temporary file; ordinary file-system operations are used to transfer pages in and out of virtual memory. This approach to paging simplifies the mechanism for migrating a process: the process's current node writes out any dirty pages to the process's swap file(s), then sends descriptors for the swap files to the kernel of the new workstation. The address space of the process is then demand-paged into the new workstation as the process executes.

However, aside from operations related to the file system, system calls in Sprite are not generally location-independent. A trivial example is the time-of-day clock. If different nodes' clocks drift relative to each other, it may not be acceptable for a migrated process to use the clock of its current node. As another example, each processor maintains one or more *environments*, each of which associates particular string values with particular string names. Sprite must take special care to ensure that a migrated process uses the same environment, regardless of where the process has been migrated. Other operations require cooperation between nodes in order to perform correctly: for example, when a process exits its parent must be notified; either the parent or the child or both might have been migrated. The following section describes in detail the mechanisms for process migration and remote execution, and it explains how the transparent execution issue was resolved by assigning each process a *home node*.

4 The Sprite Process Migration Mechanisms

4.1 Migrating a Process

A kernel migrates a process by suspending the process on the *source* node and transferring its process state, virtual address space, and open files to the *destination* node. All transfers are done using the standard kernel-to-kernel RPC mechanism. Specifically, migration involves the following steps:

1. An RPC is sent to the destination node to confirm that the process will be allowed to migrate. (When evicting a process and sending it home, the RPC confirms that the home node is accessible, but the process is automatically "allowed" to migrate. Otherwise, the kernel on the destination node may accept or reject foreign processes as it chooses.)

2. The process is interrupted, using the standard signal mechanism, to keep it from executing while it is being migrated. If the process is running in user mode, it will trap into the kernel; if it is in the middle of a system call, the call will be interrupted at the next convenient place.

3. The "process state" of the process is transferred. This includes the contents of registers, user and group identifications, signal handling information, and the home node and process identifier of the migrating process.

4. The virtual address space is transferred. Any dirty pages are sent to a file server, then the page tables and descriptors for the corresponding swap files are sent to the destination node. This may require a large number of RPCs depending on the volume of dirty pages and the size of the page tables that are sent. Also, if the dirty pages do not fit in the file server's cache, some delays will ensue as pages are written to disk.

5. Descriptors for the process's open files and current working directory are encapsulated and transferred. When writable open files are transferred, the file system cache consistency protocol may cause files to be written back to file servers.

6. An RPC is sent to conclude the migration and permit the migrated process to resume execution on the destination node.

7. Finally, the process resumes on the destination node. The process's address space is demand-paged onto the destination node as pages are referenced.

4.2 Transparent Remote Execution

No matter where a process executes, it must behave as if it were running on the same workstation throughout its lifetime. Any actions that depend on the location of the process are performed as if the process were executing on its *home node*. The home node is similar to the *origin site* of LOCUS, except that a process inherits the home node of its parent: in particular, if a migrated Sprite process forks a child, the child behaves as if it had been created on the parent's home node and immediately migrated. In LOCUS, the origin site of the child process is the node on which the process is created, regardless of the origin site of the parent. The difference between Sprite and LOCUS arises primarily from naming considerations: since LOCUS has global naming, the origin site is used primarily for mapping a process's identifier to its current location, and for efficiency one desires a child's origin site to be its current host. In Sprite, the home node must be inherited to ensure consistent naming and synchronization of exiting processes; for example, processes are identified using a machine-relative process identifier, and signals may be sent only to processes that are logically executing on the same machine (even if some are migrated elsewhere).

A foreign process will execute user code and some system calls on a remote workstation, but its home node will continue to provide location-dependent services. Sprite classifies system calls in the following categories (with the number of calls in each category indicated in parentheses):

- Site-independent (38).

  The *remote* node handles calls that do not depend on the location of a process. For example, a process's virtual memory is managed entirely by the node on which it is executing. Also, the remote node performs file operations by direct communication with file servers.

- Site-dependent (24).

  The *home* node services most calls that depend on a process's location. Any operations on the environment of a process are forwarded home, as are calls to get the time of day, retrieve information about currently-executing processes, or send signals. Any calls that may take process identifiers as arguments are handled by the home node because those identifiers are relative to the home node of the process.

- Cooperative (5).

  In a few cases, the *remote* and *home* nodes must cooperate to handle a system call. For example, when a remote process exits, both workstations must clean up some state. Similarly, when a remote process creates a child, both nodes allocate state for the child. Additionally, any calls that affect process state (such as setting the effective user identifier) update the state on both hosts.

- Not migrateable (1).

  Sprite provides a system call to map portions of the kernel's address space into user memory. (The X window server uses this call to access the display.) It is not possible for a remote process to map memory from the kernel on its home node, so any process that invoked this call would have to migrate home before the call could complete.

The decision of where to handle a system call is table-driven and is based on the status of the process (local or foreign) and the particular call. If a process is running on its home node, or if the system call may be handled by the remote node directly, nothing special needs to be done. For forwarded calls, the arguments to the call are passed via RPC to the home node, where an RPC server process performs the desired operation as a *surrogate*. The process table entry of the server is marked to indicate that it is acting on behalf of a migrated process, and when the surrogate performs an operation (such as sending a signal or getting an environment variable), the user identifier and other process-specific information for the *migrated* process are used.

5 Cost of migration

The current implementation of process migration in Sprite takes approximately 480 milliseconds to migrate a "minimal" process from one Sun-2 workstation to another. (The "minimal" process we used has two dirty pages and no open files.) After accounting for the cost

Number of dirty 4K-byte heap pages	Time to migrate	Added cost per dirty page	Time to page in	Added cost per paged-in page
1	0.50s	N/A	0.00s	N/A
51	1.97s	29ms	1.04s	20ms
101	3.54s	30ms	2.12s	21ms
151	5.18s	31ms	3.22s	21ms
201	6.62s	31ms	4.32s	21ms

Table 1: Effect of dirty pages on time to migrate. The cost per dirty page is calculated by subtracting the cost of migration when only one heap page is dirty and dividing by the difference in the number of pages.

to migrate the minimal process, the time to migrate increases in proportion to the number of pages touched and file blocks written before migration (these must be written back to the server as part of migration). After migration, the time to demand-page the address space onto the new node increases in proportion to the number of pages that are referenced. Table 1 shows the times to migrate a process after it had touched a variable number of 4096-byte pages, as well as the time for it to reference each page once after being migrated; migration takes approximately 30 milliseconds per dirty page to flush the page to the file server, and then it requires about 21 milliseconds per page to fault them onto the new workstation.

Transferring open files is also costly, since RPCs are needed for each file. In addition, if the file has been written, dirty blocks may need to be flushed from the cache of the node on which the process had been executing. Each open file requires approximately 34 milliseconds of overhead for transfer, and each dirty block requires about 26 milliseconds to be flushed. Table 2 lists times to migrate a process with varying numbers of open files and dirty blocks.

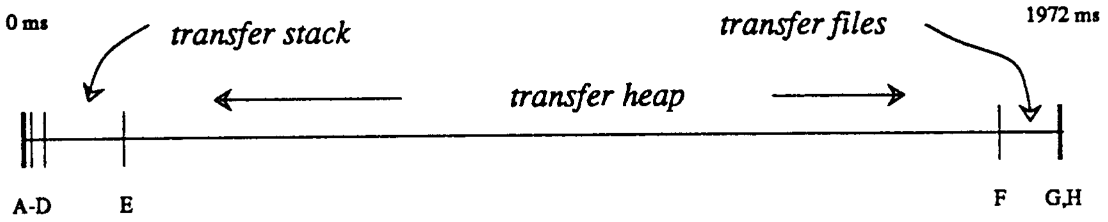
After a process has been running for a long time, the cost of migrating it will likely be dominated by the time to transfer its address space. Of the 1.97 seconds taken to migrate the process that had touched 100 pages, over 1.6 seconds were taken just to transfer its heap.

	Number of dirty 4K-byte blocks per file				
Number of open files	0	1	5	10	20
0	0.48s	N/A			
1	0.52	0.52s	0.66s	0.78s	1.04s
5	0.64	0.78	1.86	2.08	2.20
10	0.82	1.22	2.70	2.42	2.88

Table 2: Effect of open files and dirty file blocks on time to migrate. All times are the number of seconds taken to migrate a process after it has opened a number of files and written data to them. The times to migrate are not necessarily proportional to the number of blocks: when many blocks are written, some may be flushed to the server prior to migration.

The time-line for the migration of this process is given in Figure 1. It is immediately apparent that the cost of sending the stack and heap to the file server and the page tables to the remote node greatly outweighs the other costs.

These measurements support Theimer's use of *pre-copying* to reduce the amount of time during which a process is not able to execute. Nevertheless, Sprite does not pre-copy the address space for three reasons. First, Theimer was concerned about the real-time needs of his message-based system (timeouts might occur if a process were suspended for a long time during migration). In Sprite this is not a concern, since communication is managed by the system in a way that prevents these sorts of timeouts. Second, since we expect migrated processes to be non-interactive, suspending a process for several seconds will generally not be noticed by the user. Third, pre-copying reduces the time during which a process is suspended, but it may increase the total time for migration. Sprite writes each dirty page only once, so the suspension may be quite long but the total migration time is as short as possible. When a user returns to his/her workstation and its kernel evicts foreign processes, they will move off the workstation faster than they would if pre-copying were performed. Fourth, demand page-in also helps: only dirty pages need to be transferred immediately; others are not loaded



Label	Event	Duration (ms)	Total Time (ms)
A	*Issue migration request*		
	Confirm with remote node	7	7
B	*Process traps*		
	Transfer process state	12	19
C	Transfer code segment	32	51
D	Transfer stack segment	162	213
E	Transfer heap segment	1668	1881
F	Transfer files	85	1966
G	Tell remote node to resume process	6	1972
H	*Migration request completes*		

Figure 1: Time-line for migration of a process touching 50 4096-byte pages, having three open file descriptors. The dominating costs are the transferral of the stack, heap, and open files of the process. The entire operation takes approximately 2 seconds.

on the new node until referenced.

We have no specific measurements of Demos/MP or LOCUS with which to compare the performance of Sprite process migration. The cost of migration in V has been published [7], and the base cost of 80 milliseconds to transfer process state in V is greater than the cost in Sprite; however, by transferring more state the V-System makes processes less location-dependent. Sprite takes additional time to transfer file information that V does not need to transfer (due to global servers). It is difficult to compare the transfer rates for virtual address spaces, because only the maximum memory-to-memory transfer rate of the V-System is listed in [7] (333 Kbytes/second on an unloaded network, with an effective transfer rate for migration of one-half to one-third of that amount).

6 Cost of Remote Execution

The cost of forwarding calls is generally an order of magnitude greater than the time necessary to handle calls locally. A forwarded system call will pay the base RPC overhead of approximately 5 milliseconds per call [10], plus the time to encapsulate and transfer the arguments and results of the call. The simplest calls pay the greatest penalty when executed remotely, since they involve very little processing in the local case. For example, it currently takes 1 millisecond to get the value of an environment variable locally and 10.2 milliseconds for a migrated process to get the value from its home node. It takes 63 milliseconds for a local process to fork a child and wait for it to exit, but it takes 88 milliseconds for a remote process to do the same. Thus, the forwarded environment-related call is 920% slower; the forwarded *fork/wait* calls are only 40% slower.

Fortunately, forwarded calls are rare by comparison to the number of system calls that may be handled without forwarding. As shown in Section 4.2 on page 8, slightly under half of all system calls require forwarding; however, most of the forwarded calls are used less frequently than the non-forwarded calls. Table 3 lists the most frequently-invoked system calls after a workstation had been used for several compilations, directory listings, and performance benchmarks; each system call either always involves a redirection to the home node or never involves one. As the table indicates, only two of the ten most commonly invoked system calls—approximately 20% of all calls—require extra processing when invoked by a migrated process.

The most common operations for which migrated processes pay a penalty are those involving process creation and waiting for children. Process creation requires an extra RPC to the home node, to allocate a process structure corresponding to the remote child. When a migrated process exits, all resources associated with the process on its remote node are freed, and the process structure on its home node is marked as exiting; after the parent performs a wait for the child, the process structure on the home node is freed as well.

System call	Major use	Number of calls	
		Handled Remotely	Sent Home
Sig_SetHoldMask	(used primarily by *csh*)	3480	
Fs_Close	close a file	2278	
Fs_Read	read a file	1998	
Fs_IOControl	file-specific ops.	1675	
Fs_Write	write a file	922	
Proc_Wait	wait for a process to exit		460
Proc_Fork	create a child process		386
Fs_GetNewID	duplicate a file descriptor	356	
Sig_SetAction	(used primarily by *csh*)	301	
Vm_CreateVA	allocate more virtual memory	275	
All calls		11528	2548

Table 3: Summary of the most commonly-invoked system calls, as well as the total across all system calls. Many of the most common calls may be handled without forwarding them to the home node of the calling process.

Unlike LOCUS, Sprite uses a "pulling" protocol when migrated processes wait for children. While LOCUS sends a message directly to the parent when a child dies [5], Sprite uses the home node of the processes to communicate process termination information. When a migrated process waits for a child to exit, the remote kernel sends an RPC to the home node to register the parent's interest in its children. If any children have already died, their exiting information is returned; otherwise, the parent is put to sleep until a child dies. When the home node determines that a child has died, it performs an RPC to the remote node to wake up the parent, which then repeats its inquiry. Keeping exiting information on the home node avoids the necessity to buffer the information on the remote node of the parent while waiting for the parent to perform a wait. In retrospect, however, the method used by LOCUS is more efficient because it reduces the number of RPCs required for a parent to wait.

The actual penalty paid by migrated processes may be measured by comparing the time taken to perform one or more system calls by local and migrated processes running on otherwise idle workstations. Table 4 lists some measurements of the overhead of migration and remote execution. It lists the times to perform some basic system calls, as well as measurements of the costs of process creation and synchronization. For the latter measurements, the program creates a pipe, then repeatedly performs the following steps:

1. Fork a child,
2. (Optionally) migrate the child to another client,
3. Write to the pipe, and
4. Wait for the child to exit.

The child in each case waits for something to be written to the pipe, then exits. This ensures that the child cannot run to completion before its parent has the opportunity to migrate it. When run completely locally, the loop took an average of 63 milliseconds to complete. When the parent ran remotely, in which case all forking, waiting, and exiting required interaction between the home and remote nodes, the loop averaged 87 milliseconds per iteration. This is in sharp contrast to the 680 milliseconds per iteration when each child was migrated individually, and it implies that the cost of migration dominates the overhead of remote execution unless the process runs for at least several seconds.

7 Overall Performance

The discussion thus far has considered only migration between two idle workstations, but the true test of the effectiveness of the process migration facility is the net gain in throughput due to parallel remote execution. We used a simple benchmark to simulate the effect of distributing load across workstations, by compiling identical copies of the same source code in parallel; Table 5 lists measurements of the turnaround time for these compilations using Sprite and using Sun's NFS [1]. The improvement in Sprite using remote execution for even

Program	Execution times		
	File server	Client (local)	Client (migrated)
get environment variable	1.00ms	1.00ms	10.20ms
get process and user identifiers	1.00ms	1.00ms	8.20ms
fork, synchronize with pipes, wait for child to exit	62.0ms	63.2ms	87.6ms
fork, migrate child, synchronize with pipes, wait for child to exit	N/A	680ms	N/A

Table 4: Times to complete sequences of system calls. For each benchmark, times are given for the program running on a file server, locally on a diskless client, and migrated from one diskless client to another. For these sequences of system calls, the overhead of remote execution is substantial.

two compilations is significant, but the overhead in Unix of *rsh* severely degrades performance when used for processes that execute for short periods of time.

We evaluated the impact of migrated processes on the home node by measuring the CPU utilization on the home node during execution. When all processes were local, the utilization varied between 92% and 99%, but when processes were distributed across idle workstations the utilization was 78-86%; therefore, the home node is not severely impacted by servicing migrated processes. When *every* process was migrated (leaving no compilation on the home node), the utilization dropped to approximately 10-25%, with an increase of about 4% utilization per migrated process. (The total elapsed time in the latter case was consistently higher than when one compilation executed locally, thus suggesting that it is undesirable to migrate *all* CPU-intensive processes to idle nodes.)

We measured the overhead of remote execution by performing a set of sequential compilations entirely at home and entirely on a remote node. The compilations took 66.5 seconds locally and 67.1 seconds when entirely migrated; this degradation in performance is only 1% and may be attributed almost entirely to the cost of migrating the process initially.

Program	Sprite			Sun NFS		
	Execution times		Improvement	Execution times		Improvement
	local	migrated	(%)	local	*rsh*	(%)
one compilation	19s	20s	-5	27s	43s	–59
two compilations	34	22	35	40	46	–15
three compilations	50	24	52	57	49	14
four compilations	66	28	58	72	53	26

Table 5: Comparison between Sprite migration performance and Sun NFS, running on diskless Sun-2 workstations. For one compilation, the comparison is between local and remote execution; for all other cases, the completion time for local execution is compared to the completion time when one compilation is performed locally and all others are executed remotelv on separate workstations.

8 Conclusions

The relative costs of migration and remote execution suggest criteria for deciding when to migrate processes. We anticipate two common scenarios for migration: load-balancing and preemption. The first case will use an *rsh*-like mechanism, in which a small process will be migrated before executing another program. The migration would take approximately 480 milliseconds, as described in Section 5 above. The second case arises when one or more "foreign" processes are evicted. In this case migrating all the processes back to their home nodes may take several seconds, since their virtual address spaces and cached writable files will need to be transferred. If the criteria for determining when a workstation is idle are properly determined—for example, a low load average and no keyboard or mouse activity in the past 15 minutes—then a short degradation in performance while processes are evicted should be an acceptable price to pay for the ability to share idle workstations.

Processes should be executed remotely if the following criteria apply:

- there are two or more processing-intensive processes running on the workstation;

- each process is expected to run for a period of time that is much greater than the time to start a remotely-executing process (*i.e.*, at least 2-3 seconds); and
- the likelihood of being evicted from the selected remote node is small.

References

[1] *Sun's Network File System.* Sun Microsystems, 1985.

[2] D. R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 129–140, October 1983.

[3] W. Joy, R. Fabry, S. Leffler, M. McKusick, and M. Karels. *Berkeley Software Architecture Manual.* Computer Systems Research Group, Dept. of EECS, University of California, Berkeley, Berkeley, California, 4.3BSD edition, April 1986.

[4] M. N. Nelson. *Virtual Memory for the Sprite Operating System.* Technical Report UCB/CSD 86/301, Computer Science Division (EECS), University of California, Berkeley, 1986.

[5] Gerald J. Popek and Bruce J. Walker, editors. *The LOCUS Distributed System Architecture. Computer Systems Series*, The MIT Press, 1985.

[6] M. L. Powell and B. P. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 110–119, ACM, October 1983.

[7] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems.* PhD thesis, Stanford University, 1986.

[8] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 2–12, December 1985.

[9] B. Walker, G. Popek, B. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the 9th Symposium on Operating System Principles*, pages 49–70, October 1983.

[10] B. Welch. *The Sprite Remote Procedure Call System.* Technical Report UCB/CSD 86/302, Computer Science Division (EECS), University of California, Berkeley, 1986.

[11] B. Welch and J. Ousterhout. *Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.* Technical Report UCB/CSD 86/261, Computer Science Division (EECS), University of California, Berkeley, 1985.